Data-Speculation Support for a Chip Multiprocessor

Lance Hammond, Mark Willey and Kunle Olukotun

Computer Systems Laboratory
Stanford University
Stanford, CA 94305-4070
http://www-hydra.stanford.edu/

Abstract

Thread-level speculation is a technique that enables parallel execution of sequential applications on a multiprocessor. This paper describes the complete implementation of the support for thread-level speculation on the Hydra chip multiprocessor (CMP). The support consists of a number of software speculation control handlers and modifications to the shared secondary cache memory system of the CMP. This support is evaluated using five representative integer applications. Our results show that the speculative support is only able to improve performance when there is a substantial amount of medium–grained loop-level parallelism in the application. When the granularity of parallelism is too small or there is little inherent parallelism in the application, the overhead of the software handlers overwhelms any potential performance benefits from speculative-thread parallelism. Overall, thread-level speculation still appears to be a promising approach for expanding the class of applications that can be automatically parallelized, but more hardware intensive implementations for managing speculation control are required to achieve performance improvements on a wide class of integer applications.

1 Introduction

A chip multiprocessor (CMP) architecture is a high-performance and economical solution to the problem of designing microprocessors with upwards of a billion transistors. Multiprocessor architectures make it possible to design and optimize a small high-performance processor and then replicate it across the die. This architecture offers the traditional benefits of multiprocessing systems: coarse-grain loop intensive programs and multiprogramming workloads perform well. In addition, because CMPs support very low-latency communication and synchronization between the individual processors, fine grain parallel programs also perform well [8]. However, improving the performance of integer C programs presents a challenge to a CMP because these programs do not typically contain large amounts of thread-level parallelism. Even when thread-level parallelism exists it is difficult for a compiler to analyze the data dependencies between potential parallel threads and guarantee that the threads are indeed parallel.

In this paper we describe support for data speculation on memory accesses that makes the parallelization of C programs much easier. Using data speculation, a compiler and the CMP's hardware can partition any program into threads that may execute in parallel, without regard for data dependencies. Data speculation hardware mechanisms monitor memory accesses made by the parallel threads and simply restart any threads that attempt to violate true dependencies from the original program, forcing them to re-execute sequentially. These data speculation mechanisms are particularly attractive on a CMP, because they rely heavily on a high-bandwidth, low-latency interconnect between the processors in order to transmit modified data, dependency violations, and thread control synchronization quickly and efficiently.

The contributions made by this paper are a complete, detailed description of the realistic hardware and software mechanisms required to support speculative parallelism in a chip multiprocessor. We also describe a general thread creation scheme that makes it possible to exploit non-loop parallelism in addition to the loop-level parallelism exploited by previous proposals. Furthermore, our design addresses some of the realistic implementation issues left unresolved by earlier work. We present cycle-accurate evaluation results of our implementation that augment some of the theoretical limit studies presented in earlier work.

The work described in this paper is based on earlier proposals for and implementations of multiprocessors with speculative threads. Knight proposed a speculative thread architecture for mostly functional languages [6] in which hardware is used to enforce the correct execution of parallel code with side effects. The Multiscalar paradigm [1] was the first complete description and evaluation of an architecture for speculative thread parallelism. More recently, others have described compiler and hardware speculative thread support for a CMP [9, 12, 3].

The rest of this paper is structured as follows. Section 2 gives a brief overview of the basic CMP design. Section 3 gives an overview of data speculation while Sections 4 and 5 discuss our software and hardware support for data speculation and speculative threads in detail. We present our results in Section 6. Finally, we conclude in Section 7.

2 The Hydra CMP

Hydra is our design for a single-chip multiprocessor [4]. All speculation support described and evaluated in this paper has been added to this basic design. The CMP contains 4 MIPS processors, each with a pair of private data caches, attached to an integrated on-chip secondary cache using a pair of buses as depicted in Figure 1. The processors use data caches with a write-through policy to sim-

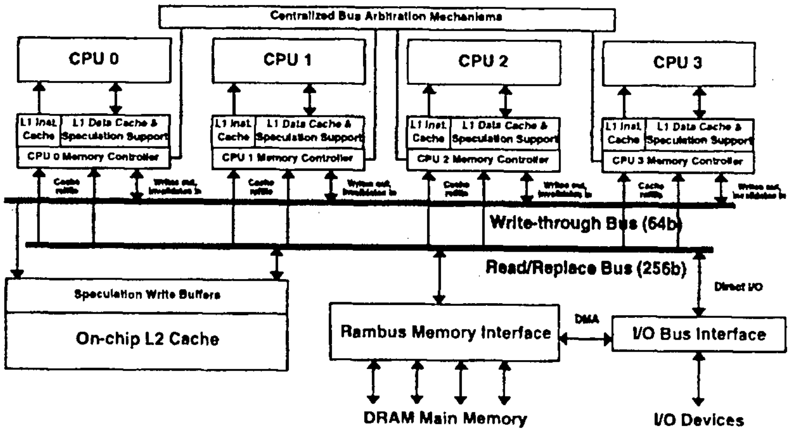

Figure 1. The main datapaths in the Hydra CMP.

plify the implementation of cache coherence. All writes propagate through to the write back secondary cache using the dedicated *write bus*. In order to ensure coherence, the other processors' data caches watch this bus — using a second set of cache tags — and invalidate lines to maintain cache coherence. Interprocessor communication is supported by processors recovering the updated version of the line from the shared secondary cache. All other on-chip communication among the caches and the external ports, such as data cache refills, are supported by the cache-line-wide *read bus*. Both buses are fully pipelined to maintain single-cycle occupancy for all accesses. Off-chip accesses are handled using dedicated main memory and I/O buses. For the applications evaluated in this paper, the bandwidth of these buses is not a performance bottleneck. A summary of the pertinent characteristics of the Hydra CMP memory system appears in Table 1.

	L1 Cache	L2 Cache	Main Memory
Configuration	Separate I & D SRAM cache pairs for each CPU	Shared, on-chip SRAM cache	Off-chip DRAM
Capacity	16KB each	2 MB	128 MB
Bus Width	64-bit connection to CPU	256-bit read bus + 64-bit write bus	32-bits of Rambus (running at the full CPU speed)
Access Time	1 CPU cycle	5 CPU cycles	at least 50 cycles
Associativity	4-way	4-way	N/A
Line Size	32 bytes	32 bytes	4 KB pages
Write Policy	Writethrough, no allocate on write	Writeback, allocate on writes	" Writeback" (virtual memory)
Inclusion	N/A	Inclusion not enforced by L2 on L1 caches	Includes all cached data

Table 1. Hydra memory hierarchy characteristics.

3 Data Speculation

Data speculation mechanisms allow instructions from a sequential instruction stream to be reordered, even in the presence of loads and stores that may be interdependent. Conventional out-of-order uniprocessors can reorder most ALU-type instructions in a RISC processor using register renaming and dynamic scheduling. However, these processors cannot reorder memory access instructions until the addresses have been calculated for all preceding stores. Only at this point will it be possible for out-of-order hardware to guarantee that a load will not be dependent upon any preceding stores. Fine-grained data speculation allows loads to be speculatively executed before these store addresses are known. If a true dependency is actually detected once the prior store addresses are known, the misspeculated load and any instructions dependent on it may be discarded and re-executed. As processor instruction windows get larger, such speculation becomes more important to allow a greater degree of out-of-order instruction processing.

Data speculation mechanisms can also facilitate the parallelization of programs for a multiprocessor. Today, programmers or compilers must carefully divide up a sequential program into separate threads that are guaranteed to be free of true dependencies through either registers or memory. This is often difficult to ensure, especially for memory references. Compilers are not able to statically disambiguate pointers in languages such as C to determine if they may be pointing to the same data structures [13]. As a result, existing compilers must assume that dependencies may be present and therefore they generate code conservatively. If a dependency may occur, the compiler either cannot divide code into threads or must insert explicit software synchronization between threads.

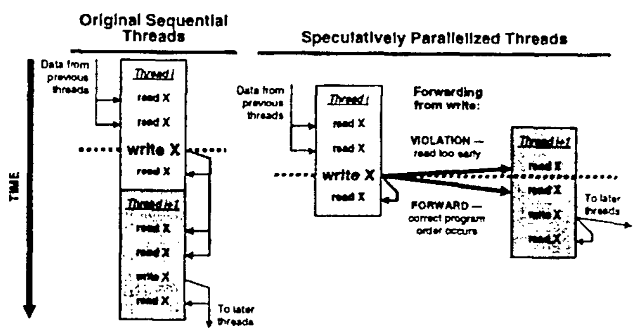

Figure 2. An example of speculative threads with data dependencies.

With thread-level data speculation, a compiler only needs to divide a sequential program into threads. These threads are given sequence numbers corresponding to the order in which they would execute sequentially, but are actually executed in parallel. The data speculation hardware ensures that true dependencies between memory accesses are always honored, even across processors, by simply backing up processors that execute a dependent load too early. Fig-

ure 2 shows how speculation hardware can use individual stores as synchronization points to detect violations or pass data between numbered threads. This mechanism allows parallelizing compilers to almost obliviously parallelize programs, since memory dependencies do not need to be explicitly grouped into a single thread or synchronized at compile time. Speculation makes the instruction windows in the parallel processors appear to be a single, large instruction window, executing a single thread made up of several disjoint sections. A compiler may parallelize as aggressively as possible, only limited by the potential performance gains from parallelization. In Section 4 we describe how we create and manage speculative threads in the Hydra CMP.

The effective memory behavior desired during speculation is summarized in Table 2 for individual accesses to an address. Writes are forwarded from earlier threads to later ones. Simultaneously, reads are recorded within each processor so that true dependence violations can be detected. Forward progress is always maintained because one thread will always execute non-speculatively, and so will be immune from violations. This *head* processor is therefore actually not speculative at all, a characteristic that can be utilized to handle exceptional situations such as calls to the operating system.

First: CPU *i*	Then: CPU *i+1*	Action
R	R	—
R	W	Rename in *i+1*
W	R	Forward written data from *i* to *i+1*
W	W	Forward occurs, but then *i+1* renames and overwrites its copy of the data

(a)

First: CPU *i+1*	Then: CPU *i*	Action
R	R	—
W	R	Rename in *i+1*
R	W	RAW Hazard: *i+1* must restart
W	W	Rename in *i+1*, so later forwarding from *i* is ignored

(b)

Table 2. Desired speculative memory behavior.

(a) shows what happens when the two processors access the location in correct program order (thread i before thread i+1), while (b) shows what happens when they access the location in reverse order (i+1 before i).

To provide the desired memory behavior, the data speculation hardware must provide:

1. A method for detecting true memory dependencies, in order to determine when a dependency has been violated.
2. A method for backing up and re-executing speculative loads and any instructions that may be dependent upon them when the load causes a violation.
3. A method for buffering any data written during a speculative region of a program so that it may be discarded when a violation occurs or permanently committed at the right time.

In Section 5, we describe how we add the memory system support for data speculation to the Hydra CMP.

4 Speculative Threads

The two existing speculative architectures take different approaches to finding speculative threads within an application. The Multiscalar architecture [11] breaks a program into a sequence of arbitrary tasks to be executed, and then allocates tasks in order around a ring of processors with direct register-to-register interconnections. While the division of a program into tasks is done at compile time, all dynamic control of the threads is performed by ring management hardware at runtime. The TLDS architecture [12], based on a chip multiprocessor, is quite different. Its hardware provides the minimum support necessary for speculation, as described in the previous section, and then all thread control is handled by software routines that are automatically added to a program at the beginning and end of speculative *epochs* by a compiler.

We use a combined hardware/software approach, similar to TLDS but with somewhat more hardware support, to divide programs into threads and then to distribute the resulting threads among the processors in the chip multiprocessor. The hardware support is a speculation coprocessor which helps execute a set of software speculation exception handlers. The extra hardware support decreases the software overheads relative to the TLDS approach and our hardware/software approach increases flexibility and decreases hardware overheads relative to the Multiscalar approach. The exception handlers divide applications into parallel threads using two techniques. First, subroutine calls cause a *fork* to occur. Afterwards, the original processor executes the subroutine, while a checkpoint of the processor state is handed to another processor so that it may attempt to execute the code following the subroutine call speculatively. Second, specially marked loops may have their loop iterations distributed among the processors. Basic compiler support for both of these techniques can be achieved without significant changes to existing compilers.

4.1 Subroutine Threads

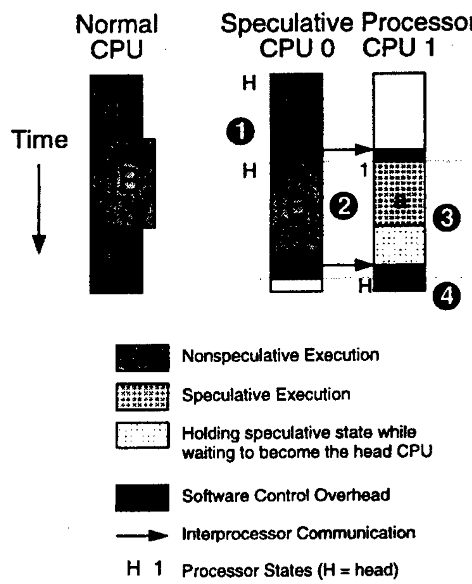

Figure 3. Subroutine fork and return.

B is a subroutine called within the *A/a* routine.

1. The call is intercepted during normal execution and the *a* thread is sent out to CPU 1, along with a newly created RPB containing its starting state and the guess for the return value of *B*.
2. The original caller continues by executing the *B* subroutine, staying the head processor as this happens.
3. Meanwhile, CPU 1 picks up the *a* thread, the caller's continuation code, and executes it speculatively. Upon completing this speculative thread, it must wait to become the head processor. During both the execution and the waiting time, its speculation mechanisms watch stores from *B* to ensure that no true dependencies between the threads are violated. The *a* thread is restarted *immediately* when such a violation is detected.
4. Upon becoming the head, CPU 1 completes and returns (or restarts and re-executes the a thread if the original return value prediction was wrong).

Subroutine speculation is controlled using a linked list of active threads ordered from least-to-most speculative and maintained by the speculation support software. When a thread is created, it is inserted into this active list. The head processor is always running the thread at the beginning of the active list, while more speculative processors try to execute the subsequent three threads from the list. Speculation is initiated with a fork message that is sent to other processors when a subroutine call is detected. Figure 3 shows the overall sequence of actions in a typical fork.

When a subroutine call is detected, several steps must occur during the actual forking operation:

1. The processor allocates a *register passing buffer* (RPB) for the thread it is creating by allocating one from the free buffer list maintained by the speculation control support software. Since our design does not incorporate direct interconnections between the processors, a buffer in memory is necessary to temporarily hold a processor's registers during the register passing communication from processor to processor. In addition, since these registers may need to be reloaded if a thread is restarted following any sort of speculation violation, it makes sense to allocate a buffer once that can hold a thread's starting (or restarting) state throughout the thread's lifetime.
2. The new buffer is filled with all registers that may be saved across subroutines (9 integer and 12 floating point using standard MIPS software conventions), the current global and stack pointers, the PC following the subroutine call, and a prediction of the subroutine's return value. For this paper, we used the simple *repeat last return value* prediction mechanism used in [10]. While more complex schemes are possible, this technique works well because most functions tend to either return the same thing continuously (void functions and functions that only return error values are good examples), or they are completely unpredictable, and therefore should not be selected for speculative execution at all. These unpredictable functions are pruned off and marked as *unpredictable* after a few mispredictions have been detected.

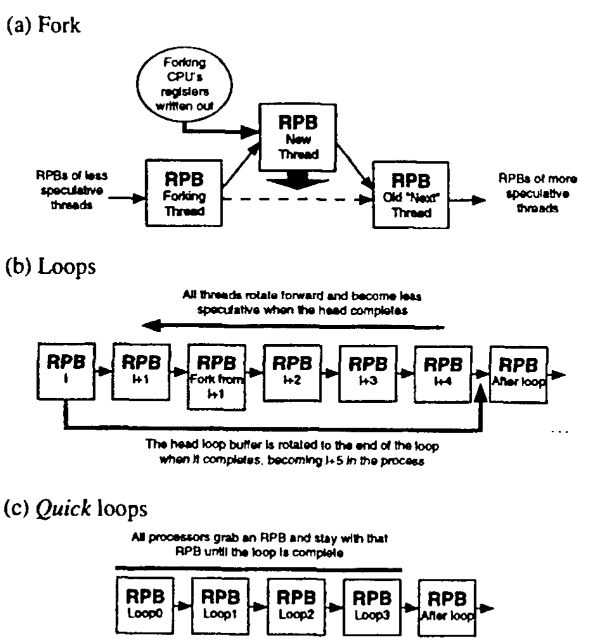

Figure 4. Managing Register Pass Buffers (RPBs).

3. The new buffer is then inserted into the list of active buffers, immediately after the current processor's, as depicted in Figure 4(a). This allows the list of active RPBs to act as the active thread list, since any *child* thread created will always be the next-most-speculative thread. The thread list must be maintained in memory for two reasons. First, any thread may be assigned to any processor over the course of its lifetime, so it is necessary to keep the thread list in a central location that all may access. Second, since there are frequently more threads than processors, it is convenient to simply leave the RPBs for these extra threads lying in memory at the end of the active list until a processor can be assigned to them.
4. The processor finishes by notifying a free processor (or, if no free processors are available, the most speculative running processor will drop its thread and pick up the new one) that it should load the registers in the newly created RPB and continue working on the code after the subroutine call.

These steps are currently performed by an exception handler that is executed when a subroutine call is detected, so that we could use commercially available compilers to compile our benchmarks. While we have vectored exceptions for speculation that avoid the normal OS exception overhead, inlining the forking code would definitely be more efficient, since only the live registers would need to be saved in the RPB. At the processor receiving the fork, another vectored exception handler gets a pointer to the new buffer from the active list, reads in the contents of the buffer into its registers, and starts executing the continuation code following the procedure call. Due to the overhead inherent in allocating a new buffer and then saving, communicating, and loading most of a processors' registers, very short subroutines are marked *unpredictable* by the return value prediction mechanism the first time they are executed so that they will not be considered for speculation on subsequent invocations.

When a subroutine completes after forking off its continuation code, it returns to the speculation support software, which performs several more steps to complete the forked subroutine:

1. It waits until it becomes the head processor. This is necessary because the processor must maintain its dependency violation detection buffers for this thread until after it becomes the head, since it may be restarted by dependence violations up until this point.
2. The actual return value of the subroutine is compared with the one predicted during the last fork. If a misprediction is detected, the return value is corrected in the RPB allocated during the last fork, and then all of the speculative processors are restarted so that they will execute using the new, correct return value.
3. The RPB of the current thread is returned to the free list as the next thread becomes the head.
4. The old head processor becomes the most speculative processor. At this point, it checks to see if there is a fourth RPB that is not assigned to any processor in the active list. If so, it starts running the thread associated with that RPB. Otherwise, it is freed until another fork occurs.

4.2 Loop Iteration Threads

A speculative loop is preceded by a check to determine whether or not it is possible to start a speculative loop. The loop is executed normally if it is known to have poor speculative performance. However, if the loop is a good candidate for speculation, a modified version of the loop body, transformed into a self-contained function, is executed repeatedly. Loop iterations are executed on all available processors. They are distributed among processors so that when

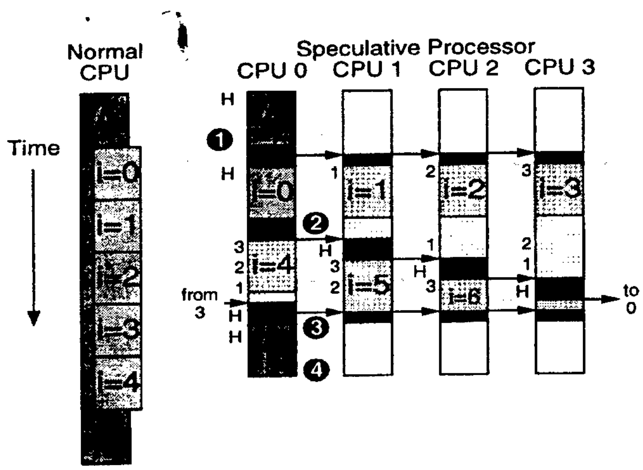

Figure 5. A simple example of a speculatively executed loop.

1. A loop is started — all processors respond, and start executing loop iterations in the order of their current state. The initiating processor also gets an iteration as it completes the *A* thread.
2. When the head processor (speculative state 0) completes a loop iteration, it notifies the other processors and starts a new loop iteration while its buffers are committed to permanent memory. This message causes the speculative states of the looping processors to shuffle. Each is decremented by 1, except the head, which now becomes the most speculative of the looping processors as it starts a new loop iteration.
3. When an iteration completes after detecting the end-of-loop condition, it sends a termination message out to all of the processors. All other processors will now be running iterations "beyond the end of the loop" that shouldn't actually execute, so they are simply cancelled, and are freed to execute any threads after the end of the loop.
4. The terminating processor picks up the *a* thread, immediately following the loop, and completes it.

loop iteration i, running on the head processor, completes, that iteration's results are committed to memory and the processor starts running the next loop iteration that has not yet been allocated to a processor, usually $i+4$, becoming the most speculative processor in the process. Meanwhile, the $i+1$ iteration becomes the head iteration and is allowed to repeat the cycle. This pattern continues until one of the loop iterations detects a loop terminating condition, and notifies the speculation system. When this processor becomes the head, all processors executing loop speculation are cancelled and execution returns to normal. A simple example of this execution sequence is depicted in Figure 5.

Two different sets of control code are used for executing loops. When starting a large loop, in which the forking of subroutines within a loop is desirable, a circle of RPBs pointing to the loop body subroutine is inserted into the active thread list when the loop is started (Figure 4(b)). Subsequently, when a loop iteration completes on the head thread, its RPB is recycled to the end of the loop, as the figure indicates. Aside from the RPB recycling and the fact that fewer registers must be saved and restored when starting a loop subroutine, the system works much like it does with procedure forks. Since the active RPB list works the same at all times, this model allows speculative thread forks from within a loop or even a loop within a loop to work correctly. However, a loop within a loop is impractical, even it works correctly, because enough loop RPBs are always inserted into the active list when any loop is started so that all processors will always be working on the innermost loop or subroutines inside of it. Hence, RPBs from outer loop iterations will always be far enough back on the active list that they will never execute until the inner loop completes. As a result of this processor allocation scheme, if nested loops are encountered, we must choose which loop is the best choice for speculative execution. Only a system with a very large number of processors could practically consider dividing up the free processors among several different parallel or nested loops in order to run speculative iterations from more than one at a time. The second set of control code is faster, but less flexible. For loops that do not contain subroutines that need to be forked, this *quick* set of routines allocates a set of four RPBs for the loop, one per processor, and then locks each processor into an RPB (Figure 4(c)). The overhead of the control routines associated with these loops is much lower because it does not have to manipulate the active RPB list after every loop iteration to perform RPB recycling or deal with forks or nested loops inside of the loop, because these are simply executed inline. While we have not currently implemented this feature, it would be possible for a compiler to generate code that could first use the slow but sophisticated loop control routines to dynamically measure a loop's contents, including that of any inner loops, and then select the quicker routines for loops that do not need the flexibility of the full loop handler based on its measurements.

A possible problem with loop speculation is that it may increase the amount of memory traffic and the instruction count during the loop. The speculative version of the loop cannot register allocate variables that are shared across loop iterations, because the data speculation mechanisms cannot protect against true dependency violations in registers. A more complex architecture, similar to the Multiscalar architecture [11], could track dependencies between the processors' register files, but this is difficult to implement in hardware without an impact on the processor core's performance. Unlike our L2 memory system, the register files are an integral part of each processor's pipeline, and modifications to allow communication between them would likely decrease each processor's core cycle time.

4.3 Thread Size

Serious consideration must be given to the size of the threads selected using the mechanisms we have described, for the following reasons:

- Limited buffer size: Since we need to buffer state from a speculative region until it commits, threads need to be short enough to avoid filling up the buffer space allocated for data speculation too often. An occasional full buffer can be handled by simply stalling the thread that is producing too much state until it becomes the head processor, when it may continue to execute while writing directly to memory. However, if this occurs too often, performance will suffer.
- True dependencies: Excessively large threads have a higher probability of dependencies with later threads, simply because they issue more loads and stores. With more true dependencies, more violations and restarts occur.
- Restart length: A late restart on a large thread will cause much more work to be discarded, since a checkpoint of the system state is only taken at the beginning of each thread. Shorter threads result in more frequent checkpoints and thus more efficient restarts.
- Overhead: Very small threads are also inefficient, because there is inevitably some overhead incurred during thread creation and completion. Programs that are broken up into larger numbers of threads will waste more time on these overheads.

Our on-chip bus communication mechanisms between processors typically result in overheads of 10–100 cycles for most speculation operations. In order to amortize these overheads while still keeping

threads short enough to avoid the long-thread problems, threads of 300–3000 instructions are optimal.

Not all loop bodies and subroutines are in this perfect size range. Also, many of these possible threads have too many true dependencies across loop iterations or with their calling routines to ever effectively achieve speedups during speculative execution. With an infinite number of procesors, it is possible to attempt to run every loop iteration and subroutine in parallel. However, many processors would be wasted achieving negligible speedups on the nonparallel routines. Unfortunately, we only have a finite number of processors. As a result, care must be taken to allocate these processors to speculative threads that are likely to improve performance.

There are two heuristics that we use to find and prevent speculation on nonparallel threads: violation counters, to eliminate threads with many dependencies, thread timers, to eliminate threads that are too short or long, and stall timers, to find threads that are stalled too long. Once nonparallel threads are discovered, we record that they should not be speculated on in a prediction table. We currently maintain a hardware prediction table, but it would be possible to perform this entirely in software at the cost of more overhead in the thread forking routines.

4.4 Synchronization

If a compiler can identify a variable in a speculative region that is likely to cause frequent violations, it may put explicit synchronization into the code, protecting the critical region where the variable is used, to eliminate the violations caused by those regions. This synchronization mechanism is simply a busy-wait loop at the beginning of the critical region that reads a lock variable, using a load instruction that will not cause violations when a less speculative processor updates the lock (in our simulator, the MIPS load locked instruction is given these semantics during speculation, since it is not needed for normal multiprocessor synchronization while the speculation hardware is active). At the end of the critical region, a normal store instruction may be used to indicate that the lock is free to the next speculative region.

It should be noted that unlike traditional MP synchronization, speculation synchronization is only used to improve performance, and is not necessary to ensure correct code execution. As a result, it can often be avoided for many variables that would traditionally require synchronization. Instead, only the few variables that cause excessive numbers of violations are targeted for synchronization.

4.5 Support for Precise Exceptions

If a speculative thread requires operating system services through a system call or an exception, the thread is stalled until it becomes the head processor. At that time, the operating system, which is not compatible with speculative execution, may be safely entered. If a thread violates or is aborted while waiting, the operating system call or exception is simply discarded. This is critical because speculative threads frequently cause segmentation faults by dereferencing null pointers or accessing data beyond the end of arrays. These extraneous segmentation faults must be squashed because they would not occur in sequential execution.

4.6 System Level Issues

In our implementation, speculative threads can coexist with other speculative and non-speculative threads from the same process or from a completely different process. When a point in the execution of the program is reached where there are explicitly parallel threads generated by a compiler or by hand, it is possible to turn off the speculative support and just execute the threads like a traditional multiprocessor. This can be done dynamically as the program executes. Speculation can be re-enabled when a speculative region of the program is reached. A feature of our speculative thread implementation is that it is possible for the operating system to steal one or more processors from a process while it is executing a speculative region. In this case the speculative control mechanisms release the most speculative processors. These processors can be used to run other speculative or non-speculative threads from another process.

4.7 The Speculation Control Coprocessor

The hardware-software interface used to control speculative threads is implemented using the MIPS coprocessor CP2 interface. Our simple coprocessor has several hardware mechanisms for controlling speculation. A collection of small software control routines is used to operate CP2. These functions are listed in Table 3. As is noted in the table, some are invoked directly by software, while others act as exception handlers triggered by hardware events or messages from other processors in the system. CP2 maintains a table of exception vectors for speculation events, so these exception handling routines can all be started without the overhead of the operating system's normal exception dispatcher. Internally, the coprocessor uses four identical state machines to track the state of the threads executing on all processors, so that exceptions may be generated or screened correctly depending upon the overall state of the system. Finally, the coprocessor contains the timers and prediction tables used to prevent speculation on nonparallel threads and to predict return values for speculative procedure continuations.

Many state transitions are initiated by messages sent between processors during the speculation control routines. These stores are all to a special memory address used only for message passing, using normal store instructions. When another processor sees a store to this special address on the write bus, it responds by modifying its internal state, and/or triggering an exception and starting the appropriate handler.

5 Hardware Support for Data Speculation

Previous data speculative architectures have proposed several different mechanisms for handling speculative memory accesses. The first was the ARB, proposed along with the Multiscalar processor [2]. This was simply a data cache shared among all processors that had additional hardware to track speculative memory references within the cache. While a reasonable first concept, it requires a shared data cache and adds complex control logic to the data cache pipeline which has the potential to increase load latency and limit data cache bandwidth. More recently, the Multiscalar group has introduced the speculative versioning cache [3], a set of separate data caches distributed among the processor cores in the Multiscalar processor that maintain their speculative state within the caches using a complex and sophisticated writeback cache protocol. Concurrently, the TLDS researchers have proposed a similar scheme [12]. However, they chose to keep their protocol much simpler, at the expense of performance-limiting bursts of coherence bus traffic at the end of every speculative epoch and an inability to forward data from speculative iterations using normal memory references. Instead, they added a special shadow memory for critical values that require early forwarding between epochs. This places an added burden on the compiler to identify the values that need to be forwarded.

Group	Handler	Use	What it does	Overhead (instr.)
Subroutines	Fork	Exception, generated upon execution of a jump-to-subroutine (JALR) instruction	Allocates an RPB, saves necessary processor state into the RPB, and then sends a FORK command out to the most speculative (or free) processor.	~70
	Subroutine Completion	Routine returned to after a forked subroutine completes, instead of the normal return address	Waits to become the head, commits speculative data, checks the guessed return value from the last fork against the actual one, kills speculative processors if necessary, and then continues with a "Start Buffer" command to get a new thread.	~40 + full "Start Buffer"
	Start Buffer	Jumped to by completion commands or an exception triggered by a FORK or STARTLOOP	Selects one of the first 4 RPBs on the active list, deletes any old children generated by this thread during any restarted earlier executions, reads in its contents, and begins executing the subroutine or loop thread. If insufficient threads are available, the processor is freed.	Full: ~60 (loop) or ~75 (fork) Short: ~25 (loop) or ~40 (fork)
Loops	Start Speculative Loop	Called by spec_begin() in original program code	Sets up a ring of RPBs for " slow" loop operation, configuring their buffers correctly, then sends out a STARTLOOP bus command to other processors.	~75
	Start Speculative Loop (quick version)	Called by spec_begin_quick() in original program code	Sets up 4 RPBs for "quick" loop operation, configuring their buffers correctly, then sends out a STARTLOOP bus command to other processors.	~70
	End-of-Iteration / Terminate	Routine returned to after a speculative loop iteration completes	Waits to become the head, commits speculative data, moves its RPB to the end of the loop, making adjustments to it if it was used for a fork during the loop iteration, and claims the next available iteration with a COMPLETE command, or commits its speculative data, discards all of the RPBs within the entire loop, and kills all other loop iterations with a KILL command before picking up the buffer following the loop with a "Start Buffer" command.	~45 + short "Start Buffer"
	End-of-Iteration / Terminate (quick version)	Routine returned to after a speculative loop iteration completes	Same as slow version, except without RPB manipulation and adjustments during the end of every iteration.	16 (on EOI) or ~45 + short "Start Buffer" (on terminate)
Support	Violation: Initiate	Exception, initiated by a RAW hazard detection in the L1 cache speculation logic	Causes speculative data to be discarded and the iteration to be restarted, and sends out a KILL bus command to following CPUs.	~30 (20 if in a quick loop)
	Violation: Receive	Exception, initiated by KILL bus command	Causes speculative data to be discarded and the iteration to be restarted.	8 + full "Start Buffer" (11 if in a quick loop)
	Hold: Buffer Full	Exception, initiated by buffer-management hardware within each CPU, due to a full L2 buffer or the possible loss of read-bit information due to an L1 replacement	Causes the CPU to stop until it becomes the head CPU, when it is allowed to continue non-speculatively, bypassing the full buffer.	15
	Hold: Exception	Exception, initiated in place of a "normal" instruction exception or system call trap encountered during speculation	Causes the CPU to stop until it becomes the head CPU, when it is allowed to continue non-speculatively with the SYSCALL or instruction exception.	25 + OS time

Table 3. A summary of the software handlers required to support speculation

In our implementation, we took advantage of Hydra's write-through data caches and bus system to build a protocol that is easily implemented without impacting the CMP's basic cycle time, is as efficient as the SVC scheme, and only requires simple coherence protocols. To add this speculation support, several key hardware elements have been added to the existing Hydra design. A block diagram of these additions, including their interface to the control coprocessor, is shown in Figure 6. The remainder of this section describes these additions.

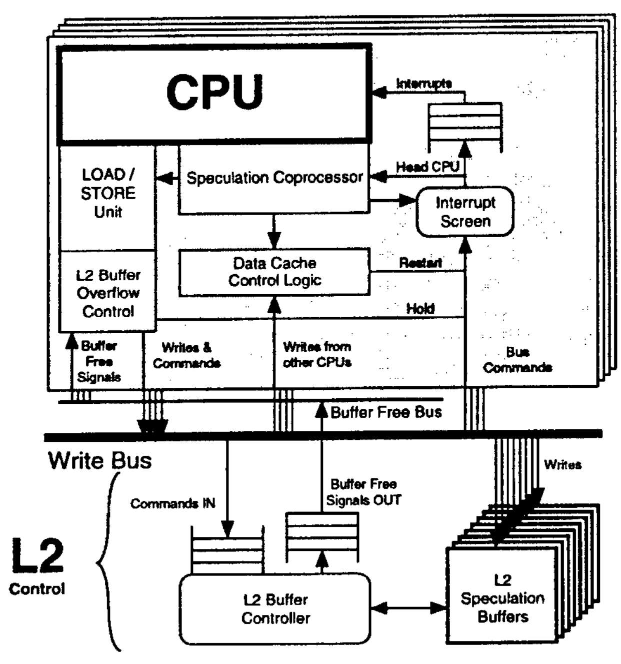

Figure 6. Hydra speculation hardware.

5.1 Data Cache Modifications

Each data cache line tag includes several additional bits to record state necessary for speculation as shown in Figure 7. The first two bits are responsible for modifying the basic cache coherence scheme that invalidates a data cache line only when a write to that line from another processor is seen on the write bus.

- Modified bit: This bit acts like a *dirty* bit in a writeback cache. If any changes are written to the line during speculation, this bit is set. These changes may come from stores by this processor or because a line is read in that includes speculative data from active secondary cache buffers. If a thread needs to be restarted on this processor, then all lines with the modified bit set are gang-invalidated at once.
- Pre-invalidate bit: This optional bit is set whenever another processor writes to the line, but is running a more speculative thread than this processor. Since writes are only propagated back to more speculative processors, we are able to safely delay invalidating the line until a different, more speculative thread is assigned to this processor. Thus, this bit acts as the opposite of the modified bit — it invalidates its cache line when the processor completes a thread. Again, all lines must be designed for gang-invalidation. If pre-invalidate bits are not included, writes from more speculative processors must invalidate the line immediately to ensure correct program execution.

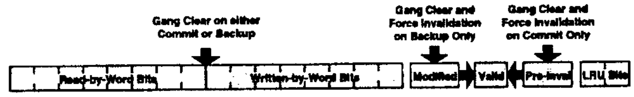

Figure 7. Data cache tag bits used for speculation.

The other two sets of bits allow the data cache to detect true dependence violations using the write bus mechanism. They must be designed to allow gang-clearing of the bits when a speculative region is either restarted or completed.

- Read bits: These bits are set whenever the processor reads from a word within the cache line, unless that word's written bit is set. If a write from a less speculative thread, seen on the write bus, hits an address in a data cache with a set read bit, then a true dependence violation has occurred between the two processors. The data cache then notifies the processor's CP2, initiating a violation exception. Subsequent stores will not activate the written bit for this line, since the potential for a violation has been established.
- Written bits: To prevent unnecessary violations, this bit or set of bits may be added to allow renaming of memory addresses used by multiple threads in different ways. If a processor writes to an entire word, then the written bit is set, indicating that this thread now has a locally generated version of the address. Subsequent loads will not set any read bit(s) for this section of the cache line, and therefore cannot cause violations.

It should be noted that all read bits set during the life of a thread must be maintained until that thread becomes the head, when it can no longer needs to detect dependencies. Even if a cache line must be removed from the cache due to a cache conflict, the line may still cause a speculation violation. Thus, if the data cache attempts to throw out a cache line with read bits set it must instead halt the processor until the thread becomes the head or is restarted. This problem can largely be eliminated by adding a small victim buffer [5] to the data cache. This victim buffer only needs to record the address of the line and the read bits in order to prevent processor halts until the victim cache is full. To simplify our current implementation, we assume that an infinite-size victim buffer, containing only read bits and addresses, is attached to each data cache.

5.2 Secondary Cache Buffers

Buffering of data stored by a speculative region to memory is handled by a set of buffers added between the write bus and the secondary cache (L2). During non-speculative execution, writes on the write bus always write their data directly into the secondary cache. During speculation, however, each processor has a secondary cache buffer assigned to it by the secondary cache buffer controller, using a simple command sent over the write bus. This buffer collects all writes made by that processor during a particular speculative thread. If the thread is restarted, then the contents of the buffer are discarded. If the thread completes successfully, then the contents are permanently written into the secondary cache. Since threads may only complete in order, the buffers therefore act as a sort of reorder buffer for memory references.

The buffers, depicted in Figure 8, consist of a set of entries that can each hold a cache line of data, a line tag, and a byte-by-byte write mask for the line. As writes are made to the buffer, entries are allocated when data is written to a cache line not present in the buffer. Once a line has been allocated, data is buffered in the appropriate location and bits in the line-by-line write mask are set to show which parts of the line have been modified.

Data may be forwarded to processors more speculative than the one assigned to a particular secondary cache buffer at any time after it has been written. When one of these later processors misses in its data cache, it requests data from the secondary, as in the normal system. However, it does not just get back data from the secondary cache. Instead, it receives a line that consists of the most recent versions of all bytes in the line. This requires priority encoders on each byte to select the newest version of each byte from among this thread's buffer, all buffers from earlier threads that have not yet drained into the secondary, and the permanent value of the byte from the secondary cache itself. The composite line is assembled and returned to the requesting processor as a single, new, and up-to-date cache line. While this prioritization and byte assembly is reasonably complex, it may be done in parallel with each secondary cache read — normally a multicycle operation already.

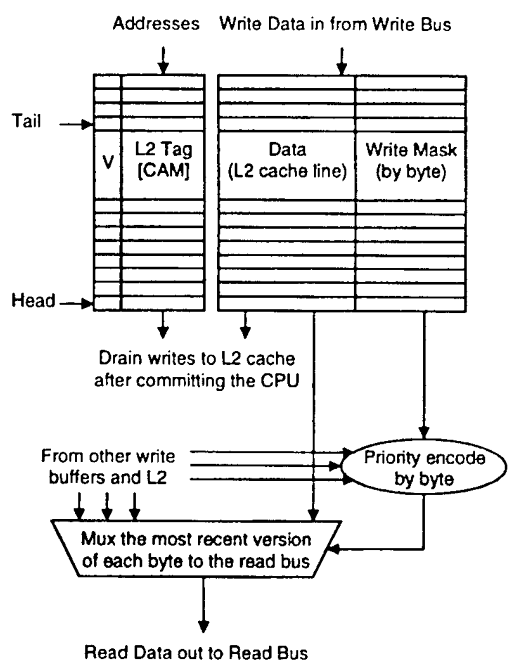

Figure 8. The secondary cache write buffers.

When a buffer needs to be drained, the processor sends out a message to the secondary cache buffer controller and the procedure is initiated. Buffers drain out entry-by-entry, only writing the bytes indicated in the write mask for that entry. Since the buffers are physically located next to the secondary cache, the buffer draining may occur on cycles when the secondary cache is free, without the use of any global chip buses. In order to allow execution to continue while buffers drain into the secondary, there are more sets of buffers than processors. Whenever a processor starts a new thread, a fresh buffer is allocated to it in order to allow its previous buffer to drain. Only in the very unlikely case that new threads are generated so quickly that all of the buffers contain data must new threads be stalled long enough to allow the oldest buffers to drain out.

Buffers may fill up during long running threads that write too much state out to memory. If these threads are not restarted, they wait until they become the head processor, write their buffers into the secondary cache, and then continue executing normally, writing directly to the secondary cache. To detect this *buffer full* problem, each processor maintains a local copy of the tags for the write buffer it is currently using. This local copy can detect buffer full conditions while the store that will overflow the buffer is executing. This store then causes an exception, much like a page fault, which allows the speculation mechanisms to handle the situation.

5.3 An Overall View of Speculative Support

To briefly illustrate how these modifications work together, Figures 9 and 10 show the operation of speculative loads and stores.

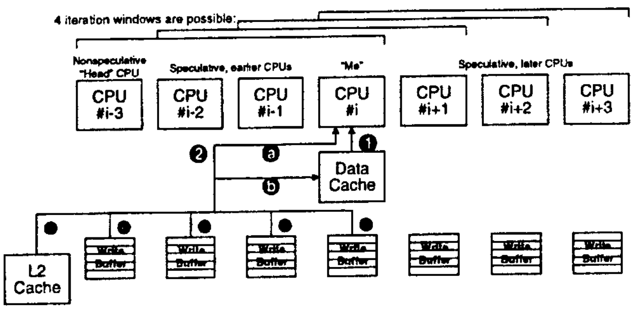

Figure 9. The operation of speculative loads.

1. A CPU first reads from its data cache. The read bit for the word is set, if the written bit for the word does not indicate that it is already a local copy.
2. In the event of an data cache miss, the L2 cache and write buffers are all checked in parallel. The newest bytes written to a line are pulled in by priority encoders on each byte, according to the indicated 1–5 priorities (1 = highest priority, 5 = lowest). This line is then returned to the CPU using the read bus. The requested word is delivered to the CPU (a), while the line is delivered to the data cache (b). The read bits for the word just read and the modified bits are set. A possible optimization would be to not set the modified bit if the line only came from the L2 cache, without any speculative additions from the buffers, but we chose not to implement this.

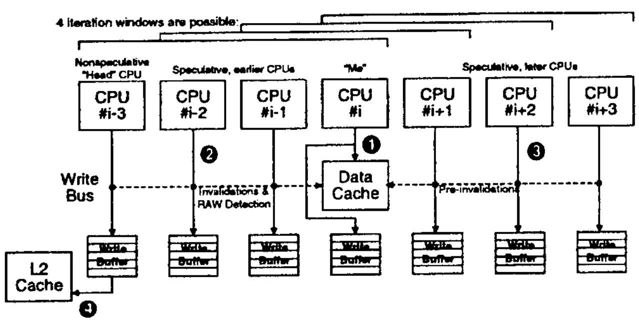

Figure 10. The operation of speculative stores.

1. On a store, each CPU writes to its data cache, if the line is present there, and its assigned write buffer, using the write bus. The modified bit of any hit lines in the data cache are set. If the read bit of the word stored to is cleared, then the written bit is set to indicate that this word is now a local copy. The data from the store is recorded in the store buffer in a newly-allocated line or included in an existing line.
2. *Earlier* CPUs invalidate data cache lines directly, if they write to a cache line present in the data cache. Also, these writes cause dependence checks. If they write to a location in the data cache or victim buffer with the read bit set, a true dependence violation has been detected, and the processor is forced to restart.
3. *Later* CPUs just cause the pre-invalidate bits in our data cache lines to be set, so that the lines will be invalidated when a new thread is allocated to this CPU.
4. When the contents of a write buffer are no longer speculative, the buffer is allowed to drain out into the L2 cache on free cycles.

6 Performance Evaluation

For our performance evaluation we use five representative integer applications written in C that are not parallelizable using conventional compiler technology. Four of the applications, `compress`, `m88ksim`, `ijpeg`, `vortex`, are from the SPEC95 benchmark suite and the fifth application, wc, is a UNIX utility. To generate speculative versions of these applications we use a simple source-to-source translator to convert the `for` and `while` loops into speculative `for` and `while` loops. The speculative source code is compiled using `cc` with `-O2` optimization running under SGI IRIX 5.3. The speculation control software was written in hand-optimized MIPS assembly language, to minimize the overhead of these critical routines as much as possible.

Our simulator models a cycle-accurate MIPS multiprocessor built from 4 simple pipelined processors, attached to a memory system that realistically models the memory delays and contention in the Hydra CMP. User code within C library functions is run under simulation, but actual operating system calls are handled by temporarily dropping from the simulator to the real machine for the duration of the call.

We present the performance results as the speedup of a four processor CMP executing a speculative application compared to one of the CMP's processors executing an optimized sequential version of the same application. The rationale for this way of presenting the performance results is that we are interested in the performance benefits of adding speculation to an existing CMP rather than a comparison of a speculative CMP with an alternative architecture.

Benchmark	Speedup	Uniprocessor Data Cache Miss Rate (%)	Speculative Data Cache Miss Rate (%)	% increase in Loads
wc	0.62	0.61	18.64	548.5
wc (w/ delay)	0.66		13.47	322.6
m88ksim	1.04	0.54	11.34	282.1
compress	1.00	4.35	15.31	-6.4
compress (w/ synchronization)	1.09		14.27	-10.9
ijpeg	1.51	0.69	5.76	63.4
vortex	0.58	1.59	13.06	38.3

Table 4. Benchmark performance summary.

Our results for the five benchmarks are summarized in Table 4 and Figure 11. The table gives speedup values from key benchmarks that we tested. The table also lists some important figures about the memory system: average miss rates for the data caches for both the non-speculative and speculative cases, and the percentage increase in load traffic when moving from a non-speculative to speculative mode of operation. The larger miss rates during speculation reflect the fact that interprocessor communication during speculation results in invalidations followed by data cache misses that then recover the new data from the L2 cache. The increased number of loads is due to a combination of running speculative control handlers, superfluous speculative memory accesses performed by speculative threads that are subsequently restarted, and the fact that speculative loops cannot register allocate actively communicated variables, increasing the number of memory reference instructions that must be generated to do the same work. However, `compress` was an anomaly, since the code that was generated by our compiler for the entire uniprocessor `compress()` function required more register saving across function calls than the small section of code within our subroutine-packaged version of the loop body used by the speculative loop mechanism.

6.1 Benchmark Analysis

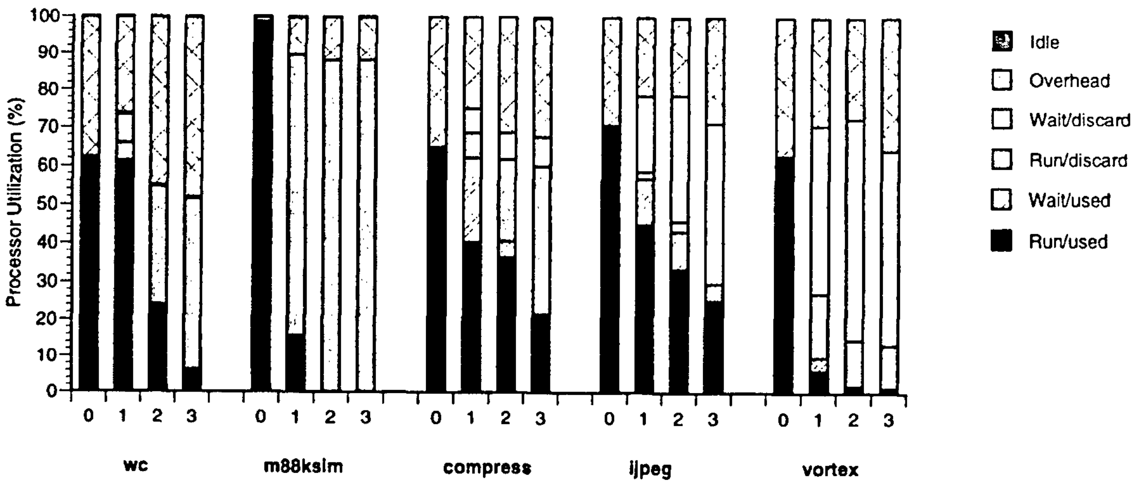

Figure 11. Processor utilization breakdown.

Our results from wc demonstrate that the software control overheads associated with our implementaion of speculation can severely limit speculation performance. The core of wc is a single loop that takes an average of only 27 cycles per iteration with fully-optimized uniprocessor code, other than the occasional iterations when the call to `getchar()` within the loop must request more characters from the OS. Even using the quick loop primitives, the speculation control software requires approximately 40% of the time on all of the processors just to handle the frequent iteration completions (on the head and #3 processor) and dependency violations (on processors 1-3), since the 10-15 instruction overhead of these operations is about half of the entire loop time! Even with this overhead, the parallelism that speculation is able to expose in the loop still allows about 40% of the system's processor time to work on the actual code. If all of this time could be exploited productively, a speedup of 1.6 could be obtained. However, an entire processor's worth of performance is lost to two factors related with interprocessor communication. First, wc has two critical loop carried-dependencies that cannot be avoided — the buffer pointer in the `getchar()` library call, and the local *in a word* variable that is used to count words. While the uniprocessor hits in its data cache when accessing these variables, a speculative processor must devote ten or more cycles to handling the data cache misses associated with this communication. Additionally, as noted previously, this communication forces the compiler to insert loads and stores to move the values to and from memory during every iteration to facilitate communication, preventing the register allocation of these commonly-used variables that may be used in the uniprocessor code. The combined effect of these two communication-related inefficiencies consumed a processor's worth of execution time on this small loop. Due to the critical nature of these memory dependencies, we also discovered that it was possible to speed up wc simply by putting a delay loop at the end of each iteration. The small delay incurred by the loop caused the iterations to pipeline more effectively, avoiding more of the overhead associated with violations at the expense of busy-waiting in the delay loop.

The loop in `m88ksim` is over two orders of magnitude larger, running for about 5000 cycles and executing an average of 4500 dynamic instructions during each iteration. With such a long loop, the overhead associated with speculation control and inter-processor communication had a minimal impact on the overall execution time. Instead, some of `m88ksim`'s global variables are read and written at locations in the loop, some inside subroutines, that severely curtail the amount of parallelism that may be exploited. The first speculative processor can use about 15% of its time usefully by overlapping the beginning of each loop iteration with the end of the previous one, but most of this time is simply spent overcoming the communication inefficiencies, limiting speedup to 3.5%. Meanwhile, the second and third processors contribute nothing, as they must work on iterations two or more ahead that cannot overlap with the head iteration at all due to true dependencies. Previous work has shown that an aggressive compiler, designed to move loads associated with receiving interprocessor communication as late as possible in each iteration and sending communicating writes as early as possible, might allow more speedup by overlapping iterations more and allowing much of the discarded time to be used effectively [12, 10], but such aggressive compiler optimizations are beyond the scope of this work.

In between wc and `m88ksim` is `compress`. The core of `compress` is fairly small loop — about 140 cycles per iteration — but large enough so that the speculation and communication overheads, while significant, do not overwhelm its execution time. Even when we left a critical loop-carried variable alone, performance was essentially equal to the uniprocessor version. However, since this single variable was such a bottleneck we were able to successfully put a synchronization point (described in Section 4.4) around it, a simple transformation that a compiler should be able to perform. By exchanging some time spent busy-waiting at the synchronization point for the longer violation-and-restart cycles that would otherwise be necessary we were able to increase iteration pipelining and achieve a 9% performance boost with this simple addition.

`Ijpeg` is an application with significant amounts of loop-level parallelism. Using our straightforward loop transformations, we were able to convert most of the loops in `ijpeg` into speculative loops that executed on all four processors. There were still occasional dependencies between loop iterations, but these did not significantly impact performance. Almost all of the discarded execution was the result of subroutine forks in the unparallelizable code (mostly in the Huffman encoding step of compression) between the

loops. These portions of the program are executed in a manner very similar to vortex, described below. This benchmark clearly indicates that our loop speculation mechanisms are able to exploit parallelism in code when that parallelism exists, even without extensive compiler optimizations. With aggressive optimizations, these results should be even better, as ijpeg is currently written so that a fairly large amount of the existing parallelism is often obscured by the existing program flow, especially during the decoding stage of the application.

Finally, our results on vortex indicate that subroutine parallelism cannot be effectively utilized by our simulator due to control software overhead and a lack of parallelism between the code in subroutines and the continuation code following them. Our simple *last value* return value prediction mechanism was able to obtain a 96.6% successful prediction accuracy when speculating on the pseudo-OOP vortex code, thanks to the large number of functions that return nothing or rarely-raised error flags. However, the severe misprediction penalty for the remaining 3.4% of the predictions — complete flushing of the system's speculative state — combined with frequent memory dependence violations originating from the side effects of the functions made parallelism virtually impossible to find. Most of the speculative processors spend their time waiting to become the head, since the wide variety of subroutines run in different threads leads to load imbalance. Each time a long subroutine becomes the head, three short ones are typically stuck waiting on the three speculative processors. Increasing the amount of parallelism exploitable would require a very sophisticated compiler that performed interprocedural optimizations to increase the distance between loads and stores that might be communicated between processors running different subroutines in parallel. It might also be forced to break up longer subroutines into smaller parts to help solve load balancing problems. It should also be noted that the overhead associated with software control of speculation is exceptionally high because vortex is parallelized only using speculative procedure continuation, which must use the full subroutine control protocol instead of the low-overhead looping protocols. As a result, the overhead is comparable to that associated with fairly small loops like wc or compress even though the subroutines are generally much larger than the loops in those benchmarks, since small subroutines are simply pruned off and avoided by our speculative thread selection mechanisms.

6.2 Memory Results

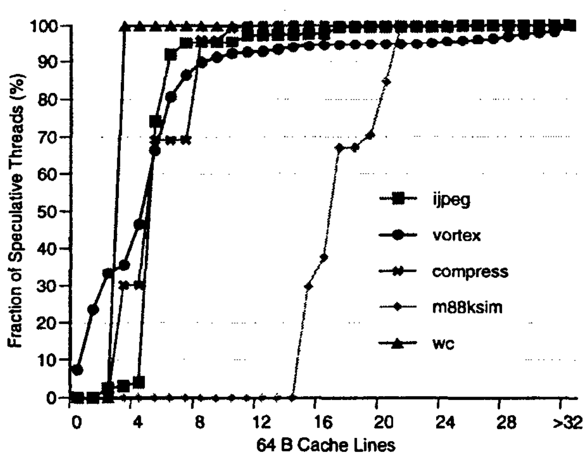

Figure 12. Speculative store buffer size requirements.

While our results indicate shortcomings in the software-based control system, the memory buffering system described in Section 5 worked very well. We found that the memory system added very little latency beyond the basic L2 cache hit time required after every communication invalidation, since it was originally designed to handle the loads of large multiprocessor FP applications. We determined that the optional pre-invalidation bits only help improve performance by 1-2%, but the hardware overhead necessary to add these bits is so small that their inclusion in the final design still makes sense. On the other hand, the write bits proved to be essential, as all of our simulations done without them resulted in the useful work done by the speculative processors dropping to nearly zero in most cases due to false violations on WAW hazards.

Figure 12 shows the numbers of 64B L2 buffers filled during each successful speculative thread. The results clearly indicate that a buffer of 24-32 lines (1.5 KB – 2 KB) per processor is sufficient to handle even fairly large loop iterations or subroutines. Even the m88ksim loop iterations would have fit in a 21-line buffer. A small number of the subroutines speculated on in vortex and ijpeg required more buffer space, but these routines were infrequent enough that simply dumping their store buffer state into the L2 cache and then completing the iteration non-speculatively after becoming the head processor would probably have little impact on performance, as most of these routines would be running on the head processor by the time they filled up a 2 KB buffer anyway.

These results indicate that the basic Hydra memory system lends itself well to speculative operation. Allowing for a fifth 2 KB buffer, so that one is free to drain while the other four accept references from processors, the system requires only about 13KB of extra on-chip memory — smaller than one of the eight existing data caches. Even allowing for a fair amount of control logic overhead, it seems reasonable to believe that the hardware we propose would not be larger than a pair of the existing data caches, which is a small fraction of the total die area.

7 Summary and Conclusions

We have demonstrated that by judicious use of hardware and software mechanisms it is possible to add data speculation capability to a CMP. Our results indicate that a data speculation system similar to ours can extract "hidden" parallelism from loops in uniprocessor code. It does this by allowing compilers to *obliviously* parallelize loops that cannot be fully analyzed for dependencies at compile time due to problems such as C pointer disambiguation. If there is parallelism, as on ijpeg, the application can speed up significantly. If there is not, as on m88ksim, the application still works, even if speculation provides no benefit. An optimizing compiler designed to arrange loads and stores to optimize communicated dependencies as much as possible among speculative threads might help this further [12]. However, our mechanisms to extract non-loop parallelism were hindered by the high software overhead of the reasonably complex control code, the load imbalance caused by running a mix of subroutines of varying sizes, and frequent memory dependencies caused by the side effects of subroutines. We feel that the results obtained using one particular speculation implementation are not sufficient to condemn the concept of subroutine continuation speculation, especially since we lacked special compiler support. However, our results clearly indicate that software control of speculation only makes sense if the control protocols are fairly simple, such as our "quick" loops, to avoid slowdowns when speculative code lacks parallelism and the speculative overhead is therefore wasted. Instead, more complex thread-generation and control algorithms clearly demand more sophisticated hardware support, such as that included in the Multiscalar architecture [11].

Our results clearly indicate that it does not make sense to design a CMP just to take advantage of speculative execution. The primary reasons for switching to a CMP are technology issues, its parallel programming performance, and its multiprogramming performance, as noted in [8]. However, speculation does not require the addition of a large amount of hardware to an existing CMP, so the cost-to-benefit ratio is reasonable enough to consider including it for the applications in which it proves helpful. As shown in [8], the individual processors in a CMP will typically be slower than a conventional wide superscalar processor of equal area while running sequential applications. Since the results in that paper indicated that a single, large superscalar processor would just be moderately faster than one of the processors in a CMP on these applications, speculation may allow a CMP to provide competitive performance on programs that have parallelism that cannot be extracted with a conventional parallelizing compiler. Optimizing compilers designed to generate code specifically for a speculative CMP might allow an even larger number of programs to benefit [10, 12]. Thus, speculation may help bridge the gap between the performance of CMPs and superscalar architectures on applications that a parallelizing compiler cannot handle. Also, the ability to flexibly deactivate speculation on processors is an advantage that should not be overlooked. When speculation proves to be unproductive, or if a truly parallel section of code is encountered, a CMP with speculation can quickly transform into a conventional SMP to run a parallel or multiprogramming workload with speedups that can greatly exceed those obtained only by exploiting parallelism within a thread, a feature that does not exist on conventional processors.

Considering our previous results from [8], it is clear that in the near future, a superscalar architecture is the best way to extract fine-grained uniprocessor parallelism from C integer program codes, given a certain amount of die area. However, it should be noted that the fine-grained threaded parallelism extracted by data speculation is orthogonal to the instruction-level parallelism extracted by superscalar ILP mechanisms. Thus, when it is possible to implement several wide-issue superscalar processors on a die together as a CMP, but impractical to simply make a larger single processor, a speculation mechanism similar to the one we propose might become a much more attractive method to extract additional performance from uniprocessor codes.

Acknowledgments

The authors wish to acknowledge J. Oplinger, D. Heine and M. Lam for discussions that led to many of the ideas that appear in this paper. This work was supported by DARPA contract DABT63-95-C-0089.

References

[1] M. Franklin and G. S. Sohi, "The expandable split window paradigm for exploiting fine-grain parallelism," *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 58–67, Gold Coast, Australia, May 1992.

[2] M. Franklin and G. Sohi, "ARB: A hardware mechanism for dynamic reordering of memory references," *IEEE Transactions on Computers*, vol. 45, no. 5, pp. 552–571, May 1996.

[3] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi, "Speculative versioning cache," *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, Las Vegas, NV, February 1998.

[4] L. Hammond and K. Olukotun, *Considerations in the Design of Hydra: a Multiprocessor-on-a-Chip Microarchitecture*, Stanford University Technical Report No. CSL-TR-98-749, Stanford University, February 1998.

[5] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *Proceedings of the 17th Annual International Symposium of Computer Architecture*, pp. 364-373, Seattle, WA, June 1990.

[6] T. Knight, "An architecture for mostly functional languages," Proceedings of the ACM Lisp and Functional Programming Conference, pp. 500–519, August 1996.

[7] M. S. Lam and R. P. Wilson, "Limits of control flow on parallelism," *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 46–57, Gold Coast, Australia, May 1992.

[8] K. Olukotun, K. Chang, L. Hammond, B. Nayfeh, and K. Wilson, "The case for a single chip multiprocessor," *Proceedings of the 7th Int. Conf. for Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pp. 2–11, Cambridge, MA 1996.

[9] J. Oplinger, D. Heine, S.-W. Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun, *Software and Hardware for Exploiting Speculative Parallelism in Multiprocessors*, Computer Systems Laboratory Technical Report CSL-TR-97-715, Stanford University, February 1997.

[10] J. Oplinger, D. Heine, M. Lam, and K. Olukotun, *In Search of Speculative Thread-Level Parallelism*, Stanford University, Computer Systems Laboratory Technical Report CSL-TR-98-765, July 1998.

[11] G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar processors," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 414–425, Ligure, Italy, June 1995

[12] J. G. Steffan and T. Mowry, "The potential for using thread-level data speculation to facilitate automatic parallelization," *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, Las Vegas, NV, February 1998.

[13] R. Wilson and M. Lam, "Efficient context-sensitive pointer analysis for C programs," *Proceedings of Prog. Lang. Design and Implementaion*, pp. 1–12, , June, 1995.